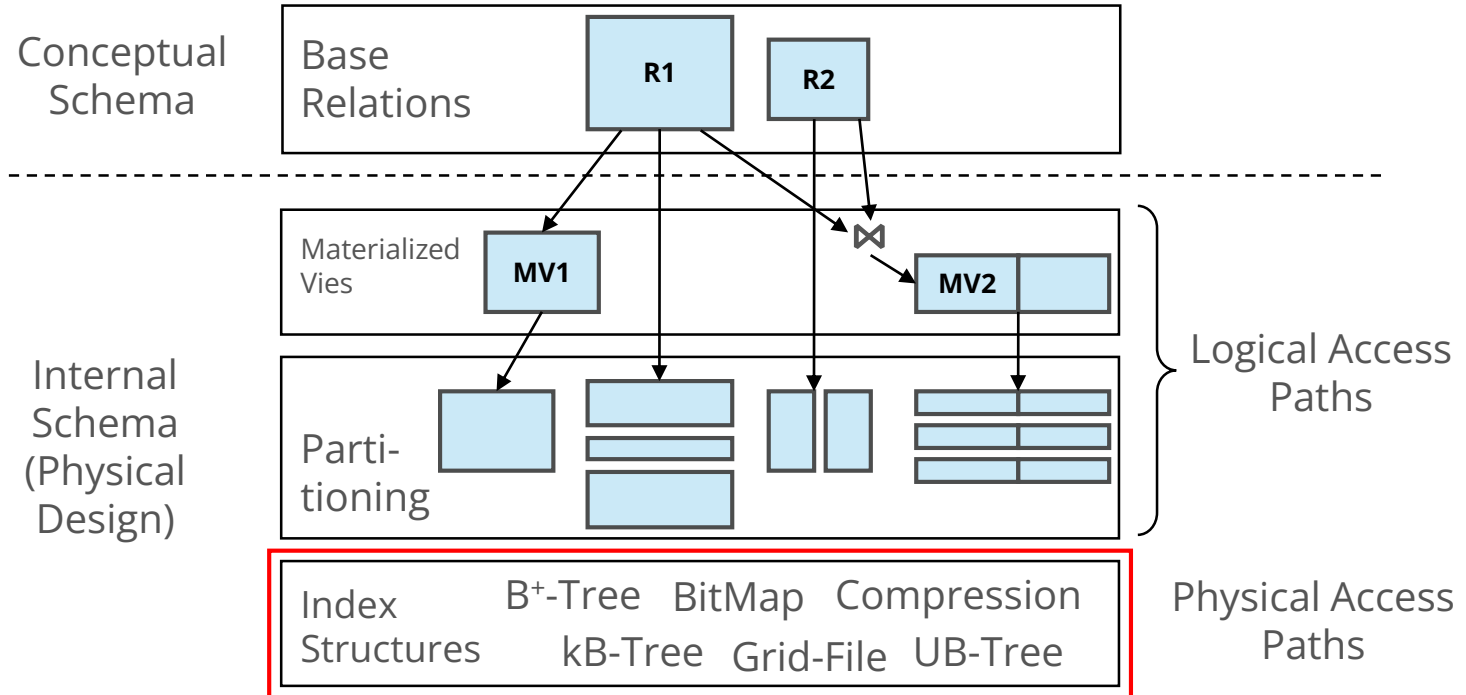


Index Structures

Scalable Data Engineering

Motivation

Overview Optimization



Motivation

Organization of Tuples

- Organized in pages
- Each tuple got an identifier (TID)

Reading Relational Data

- Entire table scan reads all pages
- Prefetching for sequential data speeds things up
- Random access usually much slower

Queries

- Often use filters
- Reading only the data necessary might save some time

Index structures allow to identify qualifying tuples.

```
select *  
from R  
where year=2021
```

Stored within
the same page

TID	year	quarter	sales
0	2020	3	10
1	2020	4	20
2	2021	1	10
3	2021	2	20
4	2021	3	30

One-dimensional Index Structures

- B-Tree, B⁺/B^{*}-Tree
- Prefix Tree (Trie)
- Data Base Cracking
- Bitmap Index

Multidimensional Index Structures

- Why one-dimensionals are not enough
- MDB-Tree
- Grid-File
- Spatial Data, R-Tree

Reusable tree

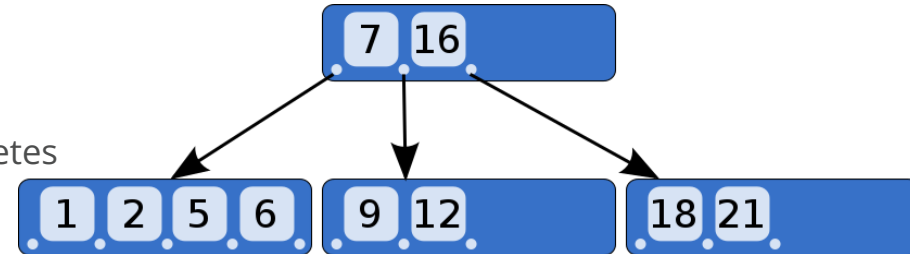
- Tree structure with several children
- Idea: The maximum size of a node corresponds exactly to the storage capacity of a page

B-Tree

- Variant of a multipath tree for mapping key values to internal record addresses (e.g., TIDs)
- Designed for use in database systems

Advantages

- Keeps keys in sorted order for sequential traversing
- Index to minimize number of disk reads
- Uses partially full blocks to speed up inserts and deletes



B-Tree Parameters

Meaning of the pointer P_i ($i = 0, 1, \dots, p$)

- P_0 points to a sub - tree with keys smaller than K_1
- P_i ($i = 1, 2, \dots, l - 1$) has a sub-tree whose key is between K_i and K_{i+1}
- P_p refers to a sub-tree with keys larger than K_p
- The pointers are not defined in the leaf nodes

Parameter k (Order of the tree)

- Is calculated from the page size
- $k = \frac{n}{2}$, i.e. $(2 \cdot k)$ Is the maximum number of entries per page

Parameter h (Height of the tree)

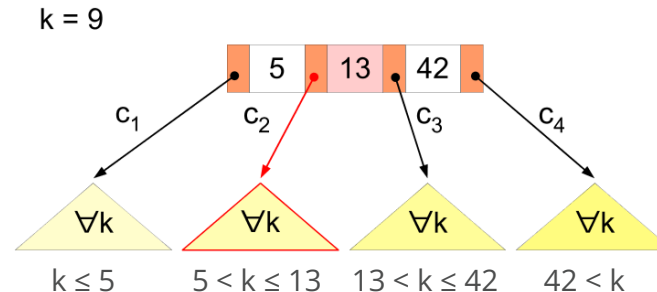
- Results from the number of stored data elements and the insertion sequence

Starting at the root node, each node is searched from left to right

- 1) if K_i matches the desired key value, the data record has been found (further records with the same key value might be located in a sub-tree to which P_{i-1} points)
- 2) if K_i is smaller than the desired value, the search will be continued in the root of the sub-tree identified by P_{i-1}
- 3) if K_i is larger than the desired value, the comparison with K_{i+1} is repeated
- 4) if K_{2k} is also smaller than the desired value, the search will be continued in the sub-tree of P_{2k}

If it's impossible to descend further into a sub-tree (2. or 4.) (leaf node):

- The search is aborted, no record with the desired key value is found



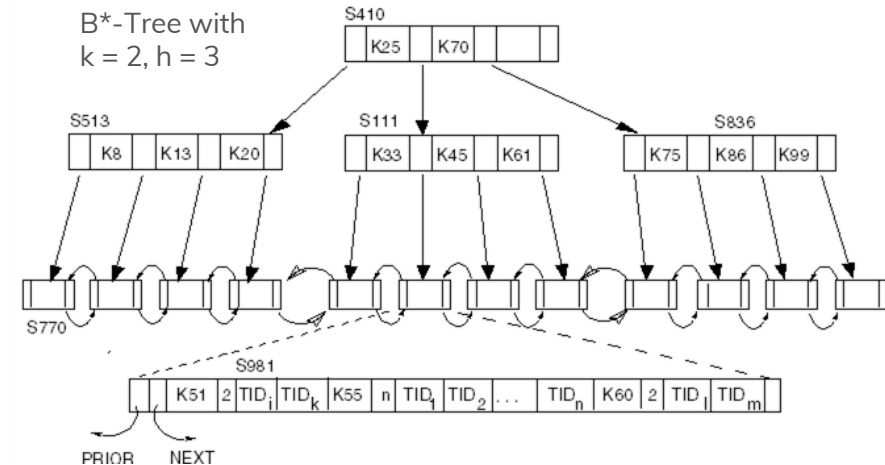
B-Trees, B⁺-Trees, and B*-Trees

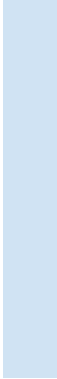
B⁺-Trees and B*-Trees

- Data is only in leaf nodes
 - Key redundancy, but higher fan-out → lower tree height, less I/O
 - Simpler delete procedure → requires only merging of nodes
- Double linked list of all leaf nodes

B*-Trees

- Modified valid node sizes:
from $[k, 2k]$ to $[4/3k, 2k]$
→ better node utilization,
but more splits/merges





Prefix Trees / Tries

Generalized Prefix Tree

Generalized Prefix Tree

- **Arbitrary data types** (byte sequences)
- **Variable prefix length k'**
- Node size: $s = 2^{k'}$ references
- Fixed maximum height $h = k/k'$
- Secondary index structure

Characteristics

- Partitioned data structure
- Order-preserving
- Update-friendly

Properties

- **Deterministic paths**
- (Worst-case complexity $O(h)$)

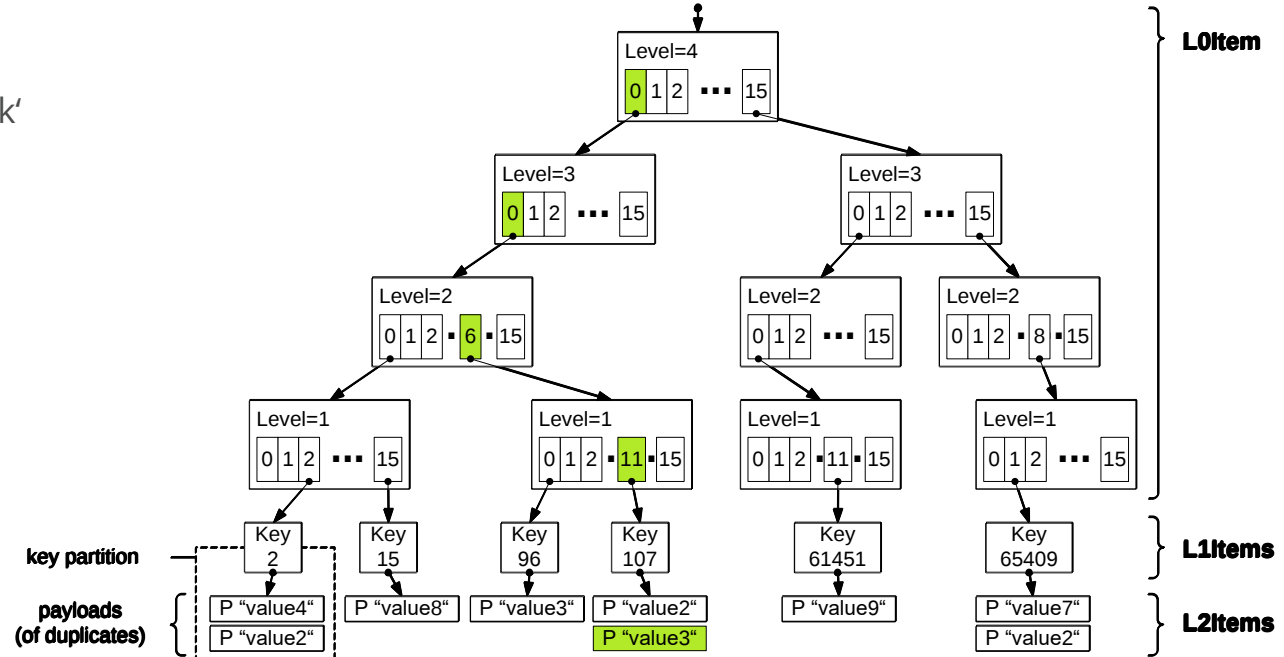
Example
 $k'=4$

INSERT key=107, payload="value3"

key = 107

0000	0000	0110	1011
0	0	6	11

Node size
($k'=4$):
128byte



Configuration Spectrum

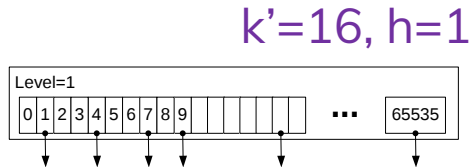
Definition (Generalized Trie)

- Variable prefix length k' , fixed maximum height $h = k/k'$
- Key observation: **read one pointer per node**
(in contrast to balanced trees)

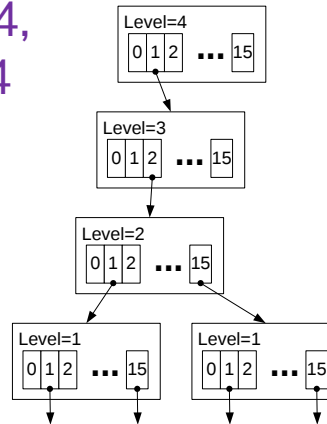
Trie Height $\rightarrow$ # Memory Transfers
 $\leftarrow$ Space Requirements $\rightarrow$ Memory Capacity / Allocation Time

Configuration Spectrum

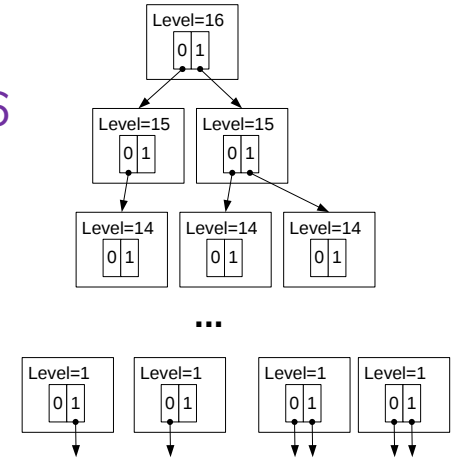
- Example $k=16$

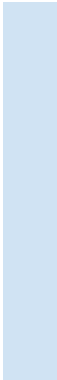


$k'=4,$
 $h=4$



$k'=1,$
 $h=16$





Data Base Cracking

The Cracker Select Operator

- **The basic select operator**
 - Scans the column
 - Returns a new column that contains qualifying values
 - **The cracker select operator**
 - Searches the cracker index
 - Physically re-organizes the pieces found
 - Updates the cracker index
 - Returns a slice of the cracker column as result
- More steps, but faster because we scan less data

Database Cracking Example (1)

Execute a range query on the unordered column A

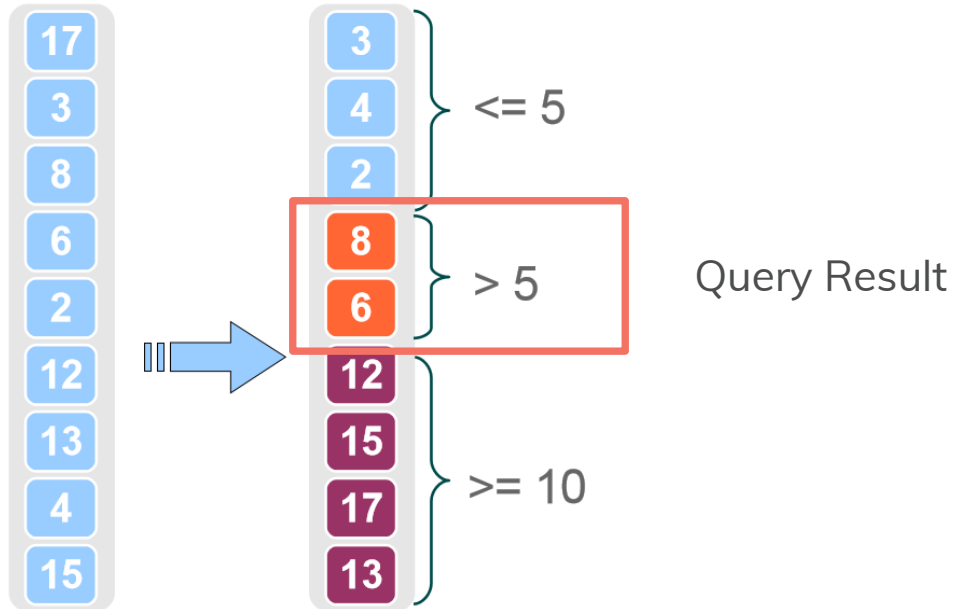
select A>5 and A<10



Database Cracking Example (2)

Build the cracker column A and create 3 slices

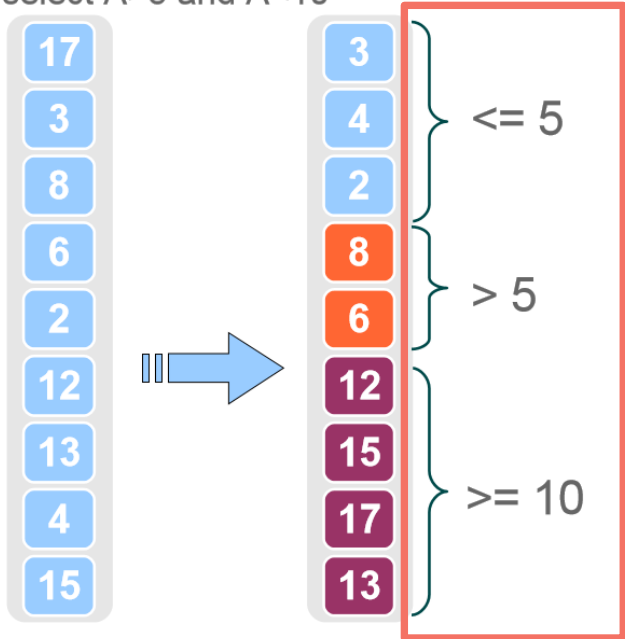
select A>5 and A<10



Database Cracking Example (3)

Build the cracker column A and create 3 slices

select A>5 and A<10



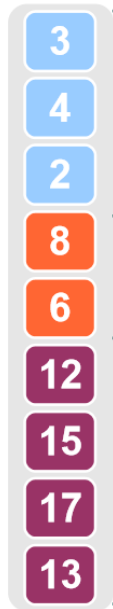
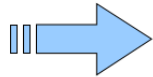
Cracker Index

- Data is partially sorted, which improves the execution time of future queries

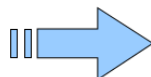
Database Cracking Example (4)

Next query needs to further sort slice 1 and 3

select $A > 5$ and $A < 10$



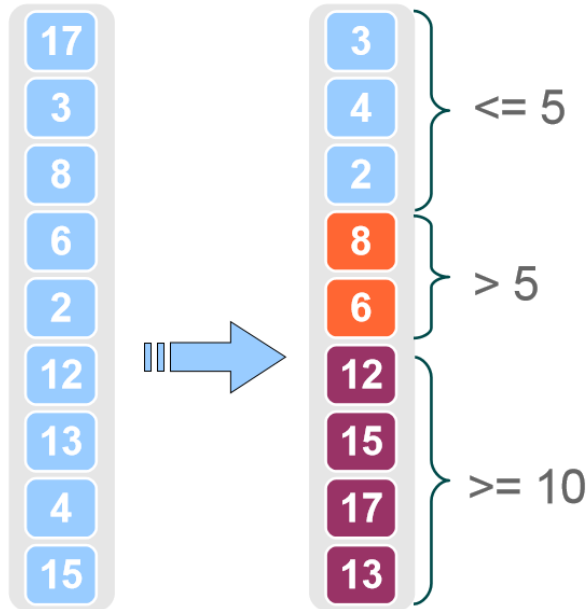
select $A > 2$ and $A < 15$



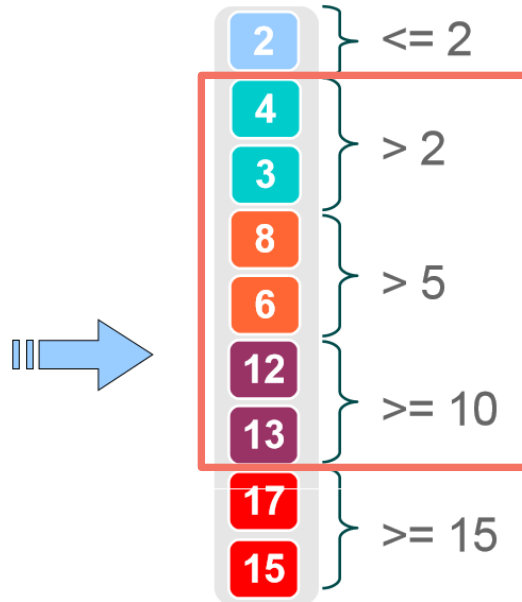
Database Cracking Example (5)

5 slices in total after this query and the cracker index became more fine-grained

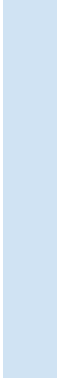
select A>5 and A<10



select A>2 and A<15



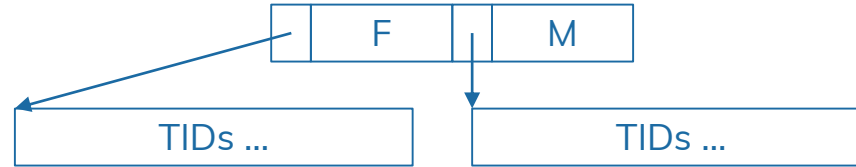
Query Result



Bitmap Index

Problem

- Example: B-tree on sex with customer table with 1,000,000 tuples results in two lists with approx. 500,000 tuples each



- Request to all "female" customers requires 500,000 individual accesses
- Full table scan is much faster

Conclusion

- B-trees (and also hashing) are useful for predicates with high "selectivity" (small proportion of expected tuples versus all tuples of a relation) Limit rate is approximately 5%.
- Higher hit rates are no longer worth the effort for index access

Bitmap Index Idea

Idea

- Already getting old ... (used since 60 years in Model 204 by Computer Corporation of America)
- Create a bitmap / bit list for each attribute expression
- Each tuple of the table is assigned a bit in the bitmap
- Bit value 1 is called the attribute value, 0 is called attribute value is not accepted
- Necessary: Continuous numbering of tuples (TIDs)

Table

Name	Sex	Region	Married
Joe	M	W	N
Anne	F	E	Y
Maik	M	E	D
Greg	M	S	N
Jenny	F	N	N
Iris	F	N	Y
...	...	...	...

Bitmap Index on
Sex

F	M
0	1
1	0
0	1
0	1
1	0
1	0
...	...

Conjunction of Bitmap Indexes

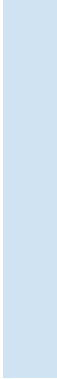
Main advantage of bitmap scripts

- Simple and efficient logical link ability
- Example: Bitmaps B1 and B2 in conjunction

Example “Married women of the 'North' region”

- Selectivity: $1/2 * 1/8 * 1/4 = 1/64$

F		N		M		*
0		0		0		0
1		0		1		0
0		0		0		0
0	&	0	&	0	=	0
1		1		0		0
1		1		1		1
...		...		...		...



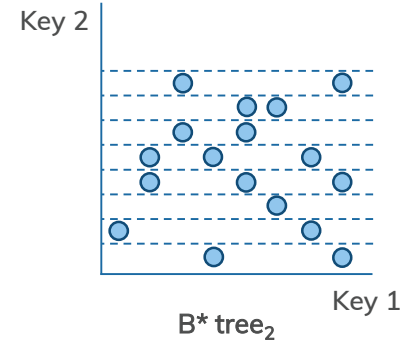
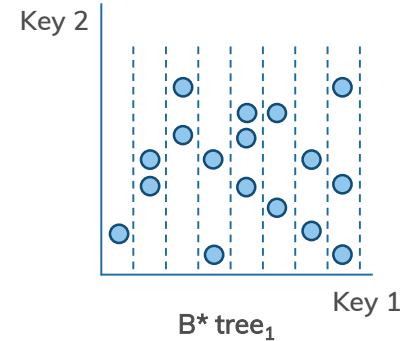
Motivation for Multidimensional Index Structures

1d Indexes (non-composite index)

- Indexing a single dimension (e.g., over B * tree)
- Example:
 - 2-dimensional search space
 - decomposition via Key1 or Key2
 - two B * trees are necessary
 - access to $Key_1 \rightarrow B^* tree_1$
 - access to $Key_2 \rightarrow B^* tree_2$

Down sides

- Creating, storing and accessing several indexes
- Might become larger than data itself



Mix RowIDs

Access to (Key₁ AND Key₂) or (Key₁ OR Key₂)

- Pointer list for values of Key₁
- Pointer list for values of Key₂
- Mix and access result tuples

Simulating multi-dimensional access with a B * Tree

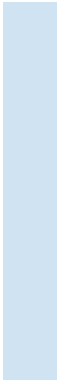
- Concatenated key values (composite index)
- Support for search operations
 - (Key₁ AND Key₂) -> OK!
 - (Key₁) -> OK for B₁
 - (Key₂) -> OK for B₂
 - (Key₁ OR Key₂) -> ?

B* tree B₁

Key ₁	Key ₂
A ₁	B ₁
A ₁	B ₂
...	...
A ₁	B _m
A ₂	B ₁
...	...
A ₂	B _m
...	...
A _n	B ₁
...	...
A _n	B _m

B* tree B₂

Key ₁	Key ₂
B ₁	A ₁
B ₂	A ₁
...	...
B _m	A ₁
B ₁	A ₂
...	...
B _m	A ₂
...	...
B ₁	A _n
...	...
B _m	A _n



Multidimensional Structures

Exact-Match-Query

- All key attributes are specified in the query
- $Q = (A_1 = a_1) \wedge (A_2 = a_2) \wedge \dots \wedge (A_k = a_k)$

Partial-Match-Query

- Only some of the key attributes are specified ($s < k$)
- E.g. `department= "K55"  $\wedge$  city = "Dresden"`

Range-Query

- For some attributes, a value interval is specified
- E.g. `department  $\geq$  "K10"  $\wedge$  department  $\leq$  "K60"`

MDB-tree: hierarchy of B-trees

Multidimensional B-trees (MDB-trees)

- Goal and idea
 - Store multidimensional keys ($a_k, \dots, a_1$) in an index structure
 - Hierarchy of trees, where each hierarchy level corresponds to an attribute.
- Principle
 - k-level hierarchy of B-trees.
 - Each hierarchy level corresponds to an attribute. Values of the attribute a_i are stored in level i ($1 \leq i \leq k$).
 - The B-trees of level i have the order m_i ; m_i depends on the length of the values of the i -th attribute.

Distribution in a B-tree

- Left sub-tree in terms of a_i (same attribute): **LOSON(a_i)**
- Right sub-tree in terms of a_i (same attribute): **HISON(a_i)**
- Point to the next attribute: **EQSON(a_i)**

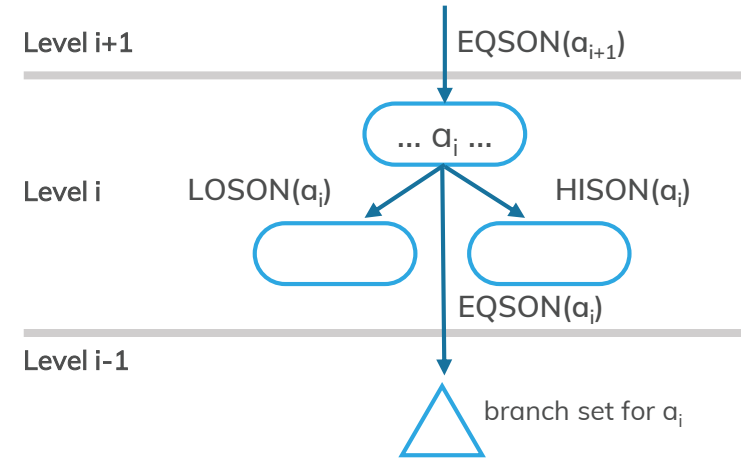
MDB-tree: structure

Link Between Levels

- Each attribute value a_i has an additional EQSON pointer
- EQSON (a_i) points to the B-tree of level $i-1$
- There, the different values belonging to keys with attribute value a_i are stored (branch set)

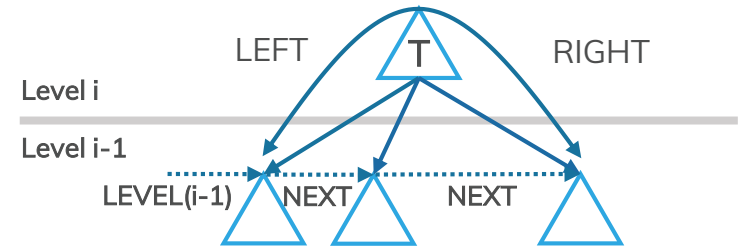
Branch Set

- Let M be the set of all keys $(a_k, \dots, a_1)$ with the prefix $(a_k, \dots, a_i)$ $i < k$
- The branch set of $(a_k, \dots, a_i)$ (brief: branch set of a_i) is the set of $(i-1)$ dimensional keys obtained from M by omitting the common prefix
- EQSON(a_i) points to a B-tree which stores the branch set of a_i



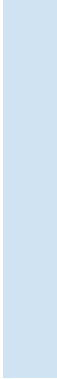
Support for query types

- Exact match queries:
-> can be answered efficiently
- Range, partial match and partial range queries:
-> require additional sequential processing at each level
- Support for range queries:
-> link root nodes of all B-trees of a level: NEXT-pointer



Support for Partial Match Queries and Partial Range

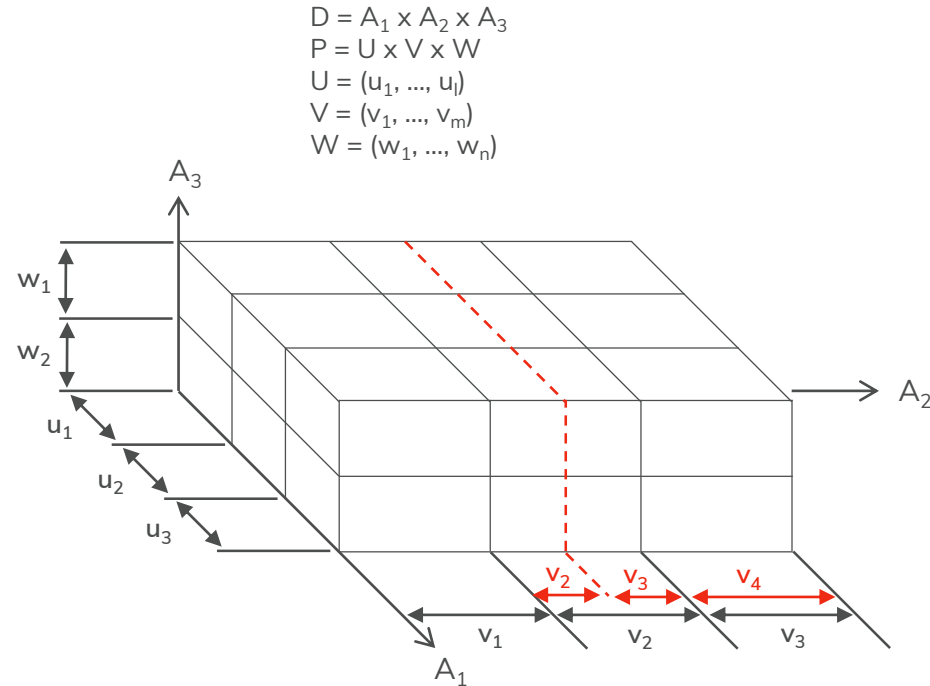
- Entry pointer for each level
 - LEVEL(i) points to the beginning of the list of linked NEXT pointers for level i
- Skip pointer from level root to the following level:
 - LEFT(T) points to the branch set of the smallest key of T
 - RIGHT(T) points to the branch set of the largest key of T



Grid File

Organizing surrounding data space by dimensional refinement

- Data space D is dynamically partitioned by an orthogonal grid, creating k -dimensional cells (grid blocks)
- The objects contained in the cells are assigned to buckets
- A distinct mapping of the cells to the buckets is defined via the grid array
- The defining characteristic of these methods is the principle of dimensional refinement, in which a section in the selected dimension is refined by a complete intersection of D .



3-dimensional data space D with partition P ;
illustration of split in interval v_2

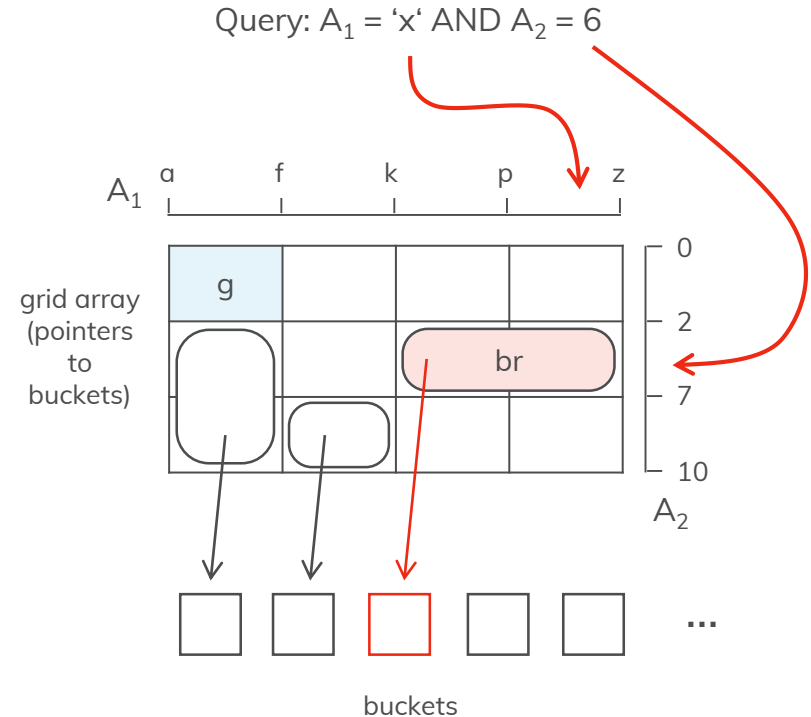
Grid-File: Grid directory

Components

- Linear scales: defines the grid on k-dimensional data space **D**
- Grid block **g**: single cell in the grid
- Grid array: a dynamic k-dimensional array whose elements (pointers to data buckets) are one-to-one correspondent with the grid blocks of the partition
- Bucket: saves the objects of one or more grid blocks (bucket region **br**)

Example

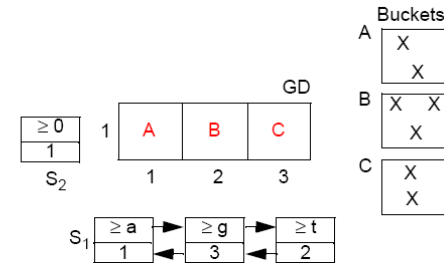
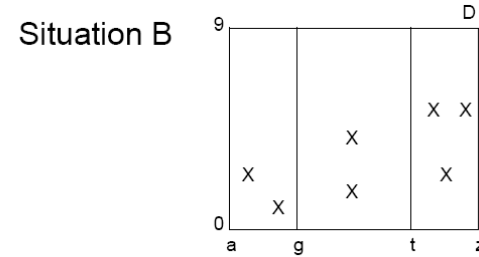
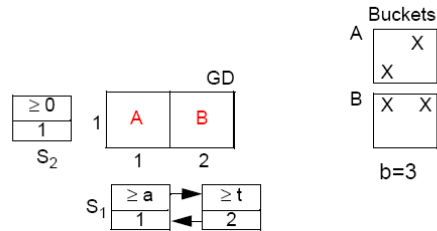
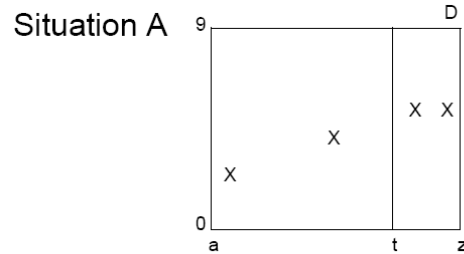
- 2-dimensional data space
- Scales A_1 and A_2 define grid blocks
- Query accesses a bucket region of two grid blocks



Grid-File: Example

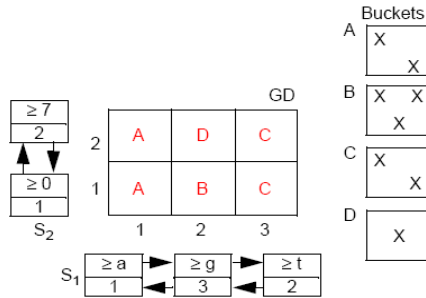
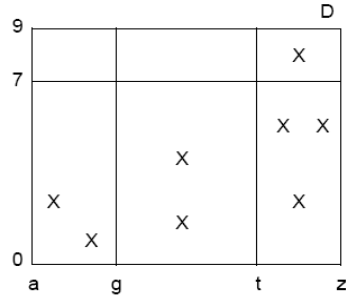
Scales as doubly linked lists

- indirection allows the grid directory to grow on the edges.
- minimize the changes to the grid directory
- stability of entries in the grid directory

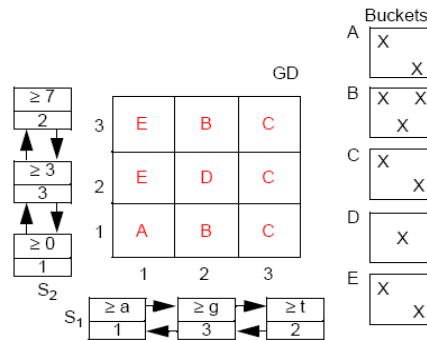
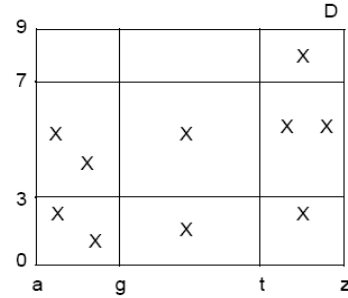


Grid-File: Example(2)

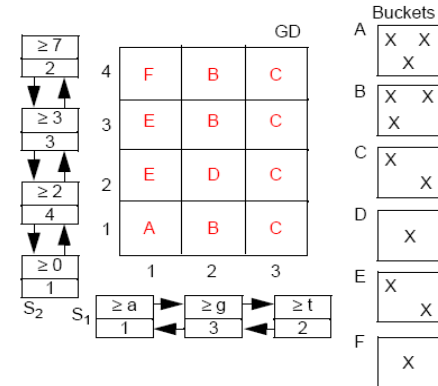
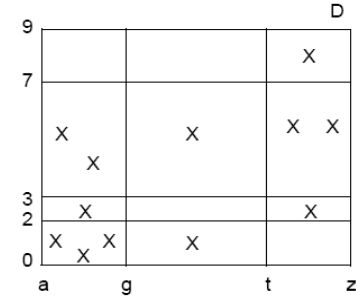
Situation C



Situation D



Situation E



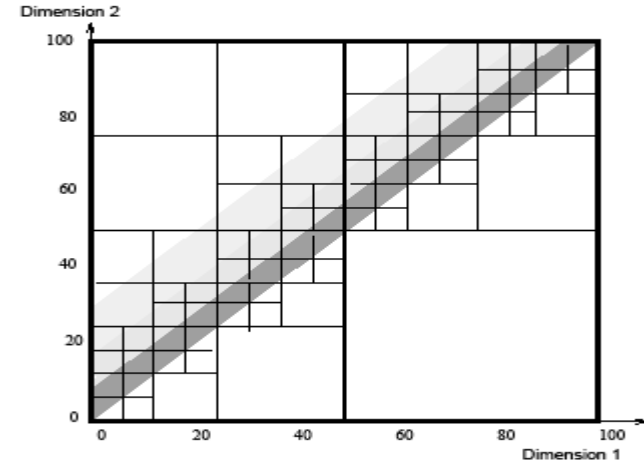
Grid-File: Degrees of freedom

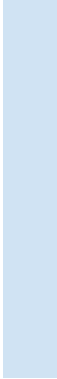
Problems

- Uneven distribution

Example: Highly Correlated Dimensions

- Order date
- Ship date
- Delivery date





Spatial Access Structures

Properties of spatial objects

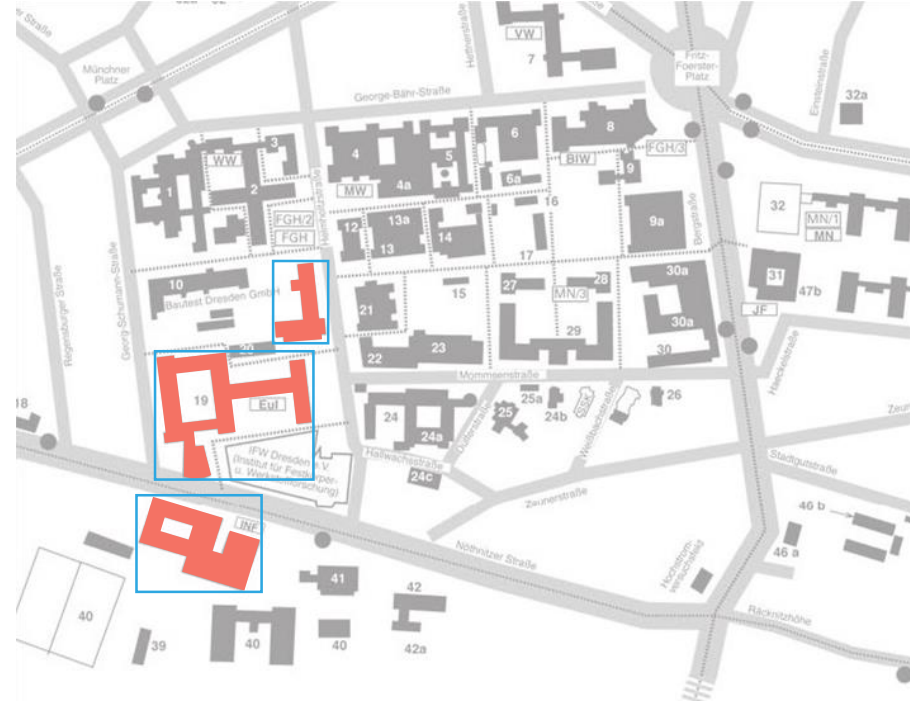
- General info such as name, texture,...
- Location and geometry (curve, polygon, ...)

Indexing of spatial objects

- accurate representation?
- object approximation via minimum bounding rectangle (MBR)

Problems

- object density and object expansion must be considered during mapping and refinement
- objects can contain other objects or overlap with each other



Query Types

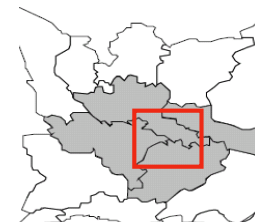
Point Query

- given a point P ; find all the geometric objects that contain P



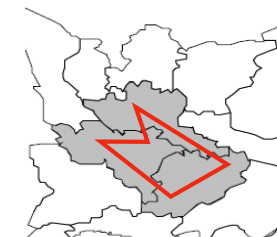
Window Query

- given a rectangle R ; find all geometric objects that intersect R



Region Query (Intersection Query)

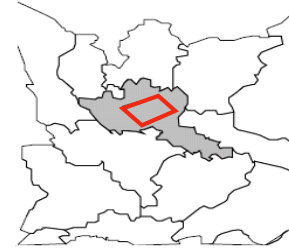
- given a polyline E ; find all the geometric objects that cut E



Query Types

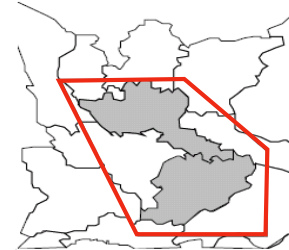
Enclosure Query

- given a polyline E; find all the geometric objects enclosing E



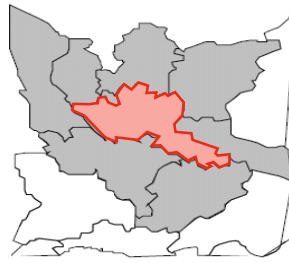
Containment Query

- given a EPL E; find all geometric objects that are completely contained in E



Adjacency Query

- given a geometric object; find all geometric objects that are adjacent to it



Goal

- Efficient management of spatial objects (points, polygons, cuboids, ...)

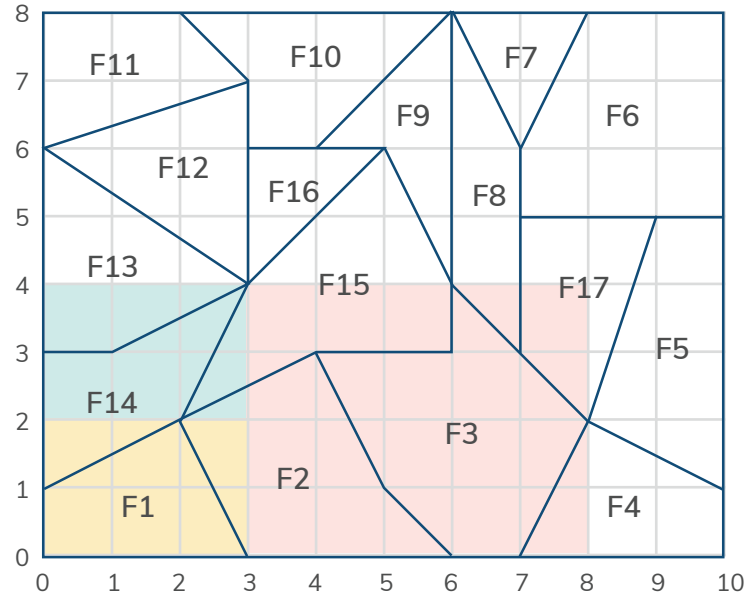
Main operations

- Point queries → find all objects that contain a given point
- Region queries → find all objects overlapping a given search window (resp. are completely contained)

Approach

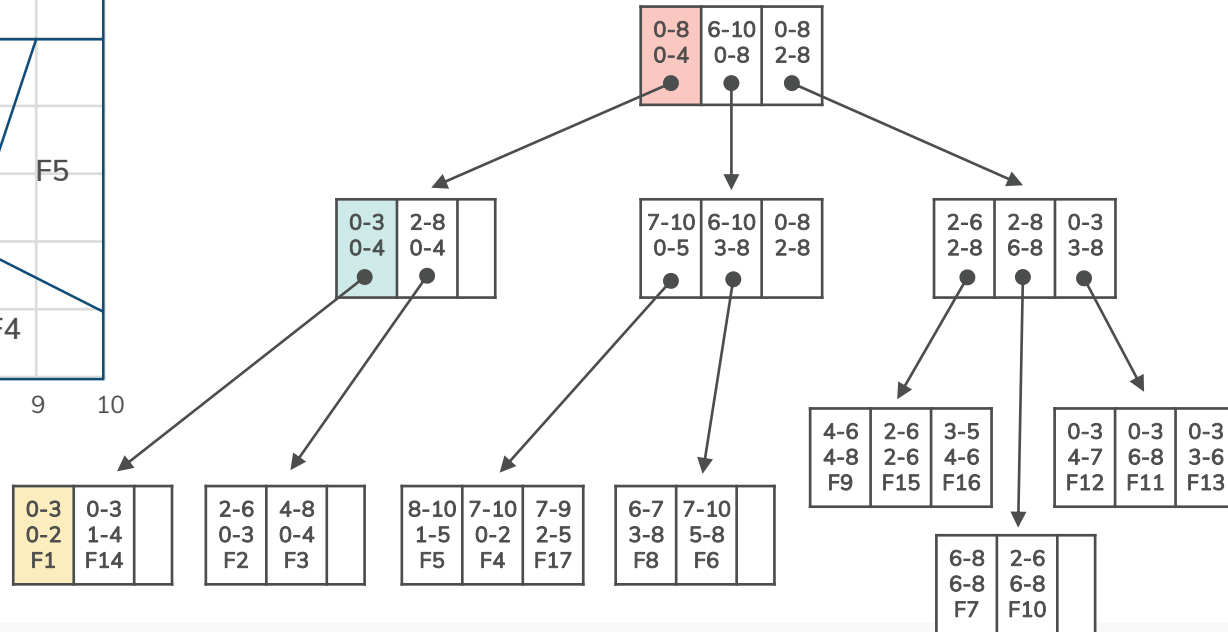
- Storage and search of axis-parallel rectangles
- Objects are represented by data rectangles and must be described with cartesian coordinates
- Representation in the R-tree is realized with minimum bounding (k-dimensional) rectangles / regions
- Search queries also refer to rectangles / regions

R-Tree: Example area objects



areas to store

corresponding R-tree



One-dimensional Index Structures

- B-Tree, B⁺/B^{*}-Tree
- Prefix Tree (Trie)
- Data Base Cracking
- Bitmap Index

Multidimensional Index Structures

- Why one-dimensionals are not enough
- MDB-Tree
- Grid-File
- Spatial Data, R-Tree